AI-ASS-7.3

2303A51018

Batch-28

Task 1: Fixing Syntax Errors

Scenario

You are reviewing a Python program where a basic function definition contains a syntax error.

Requirements

• Provide a Python function add(a, b) with a missing colon

• Use an AI tool to detect the syntax error

• Allow AI to correct the function definition

• Observe how AI explains the syntax issue

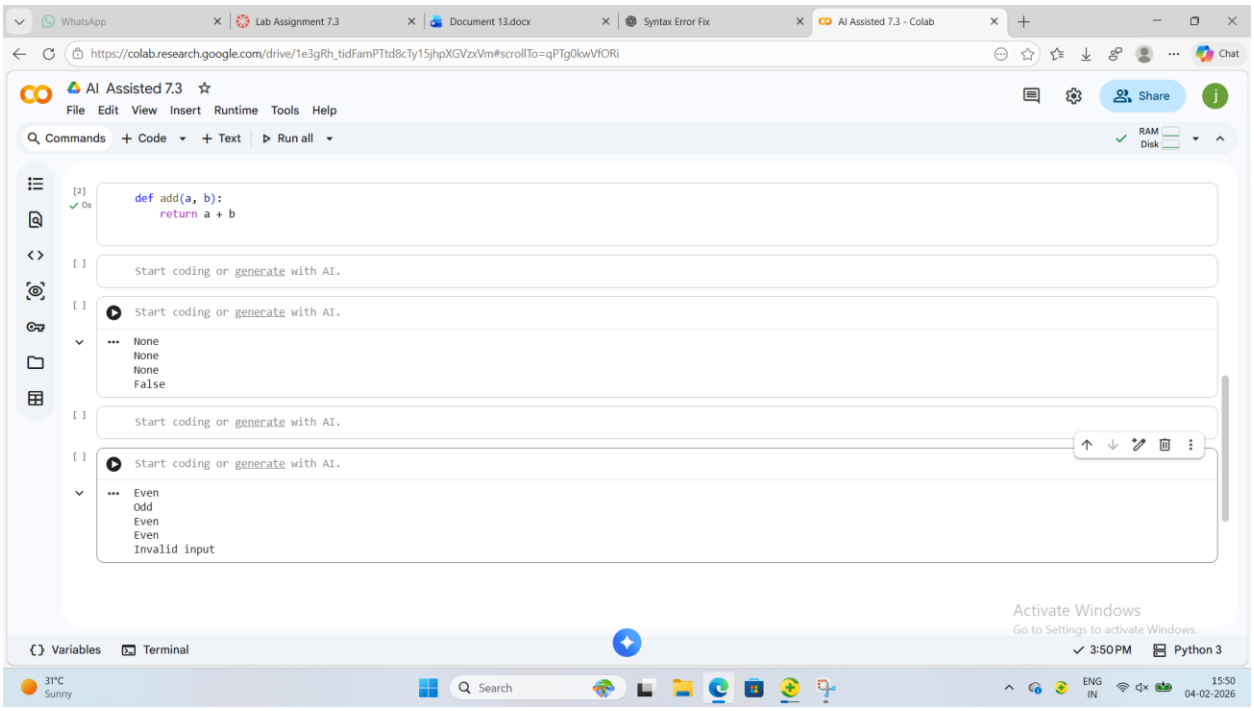
Expected Output

• Corrected function with proper syntax

• Syntax error resolved successfully

• AI-generated explanation of the fix

Code



Explanation

- The `def` keyword starts a function definition.
- The parameter list `(a, b)` must be followed by a colon `:`.
- The colon tells Python that the indented block below belongs to the function.
- Once the colon is added, the syntax error is resolved and the function works correctly.

Syntax Error Detected by AI

When this code is analyzed, Python (and the AI tool) detects a **SyntaxError**:

Reason:
In Python, every function definition **must end with a colon (`:`)**.
Without it, Python doesn't know where the function body begins.

Typical error message:

```
SyntaxError: expected ':'
```

```
Justification
```

```
Using AI to detect syntax errors is effective because:
```

- AI quickly identifies language-specific rules (like mandatory colons in Python)
- It not only corrects the code but **explains the reasoning**
- This helps beginners understand *why* the error occurred, not just how to fix it

Task 2:

Task 2: Debugging Logic Errors in Loops

Scenario

You are debugging a loop that runs infinitely due to a logical mistake.

Requirements

• Provide a loop with an increment or decrement error

• Use AI to identify the cause of infinite iteration

• Let AI fix the loop logic

• Analyze the corrected loop behavior

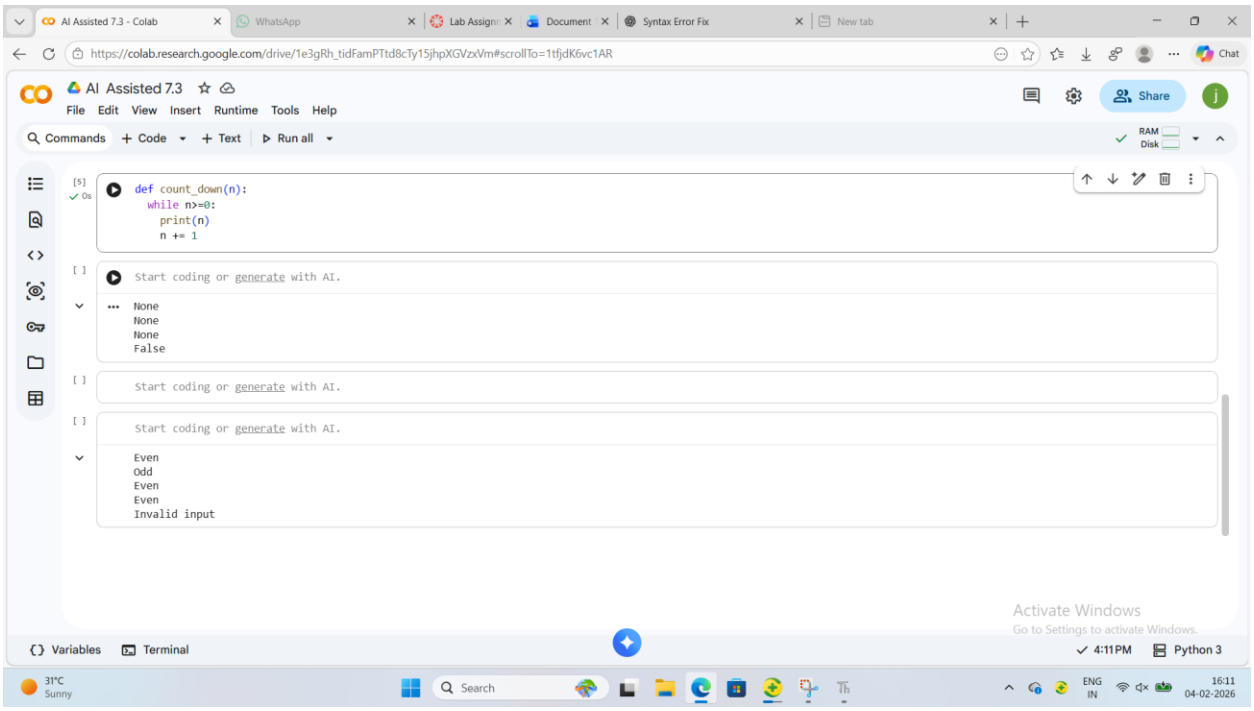
Expected Output

• Infinite loop issue resolved

• Correct increment/decrement logic applied

• AI explanation of the logic error

Code



AI Identification of the Problem

What AI notices:

- The loop condition is `i <= 5`
- Variable `i` **never changes**
- The statement `i - 1` does **not update** the value of `i`

Result:

Since `i` remains 1 forever, the condition is always true → **infinite loop**

Cause of Infinite Iteration

- `i - 1` is an expression, not an assignment
- Python evaluates it but **throws away the result**
- The loop variable is never decremented or incremented

Corrected Code (AI-Fixed Loop Logic)

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Analysis of Corrected Loop Behavior

- `i` starts at `1`
- Each iteration increases `i` by `1`
- When `i` becomes 6, the condition `i <= 5` becomes false
- Loop exits normally

Output:

```
1
2
3
4
5
```

Explanation

The infinite loop occurred because the loop control variable was never updated. By correctly incrementing the variable using `i += 1`, the loop progresses toward the exit condition and terminates as expected.

Justification

Debugging logic errors is essential because such errors do not produce syntax warnings but can cause serious runtime issues like infinite loops. In this task, the loop ran indefinitely due to the loop control variable not being updated correctly. The expression used inside the loop did not modify the variable, so the loop condition was always true.

Using AI to analyze the code helped quickly identify that the increment/decrement logic was missing. The AI not only corrected the loop by properly updating the loop variable but

also explained why the original logic failed. This demonstrates how AI tools are effective in detecting hidden logical flaws, improving code correctness, and enhancing a programmer's understanding of loop control and termination conditions.

Task 3

Task 3: Handling Runtime Errors (Division by Zero)

Scenario

A Python function crashes during execution due to a division by zero error.

Requirements

• Provide a function that performs division without validation

• Use AI to identify the runtime error

• Let AI add try-except blocks for safe execution

• Review AI's error-handling approach

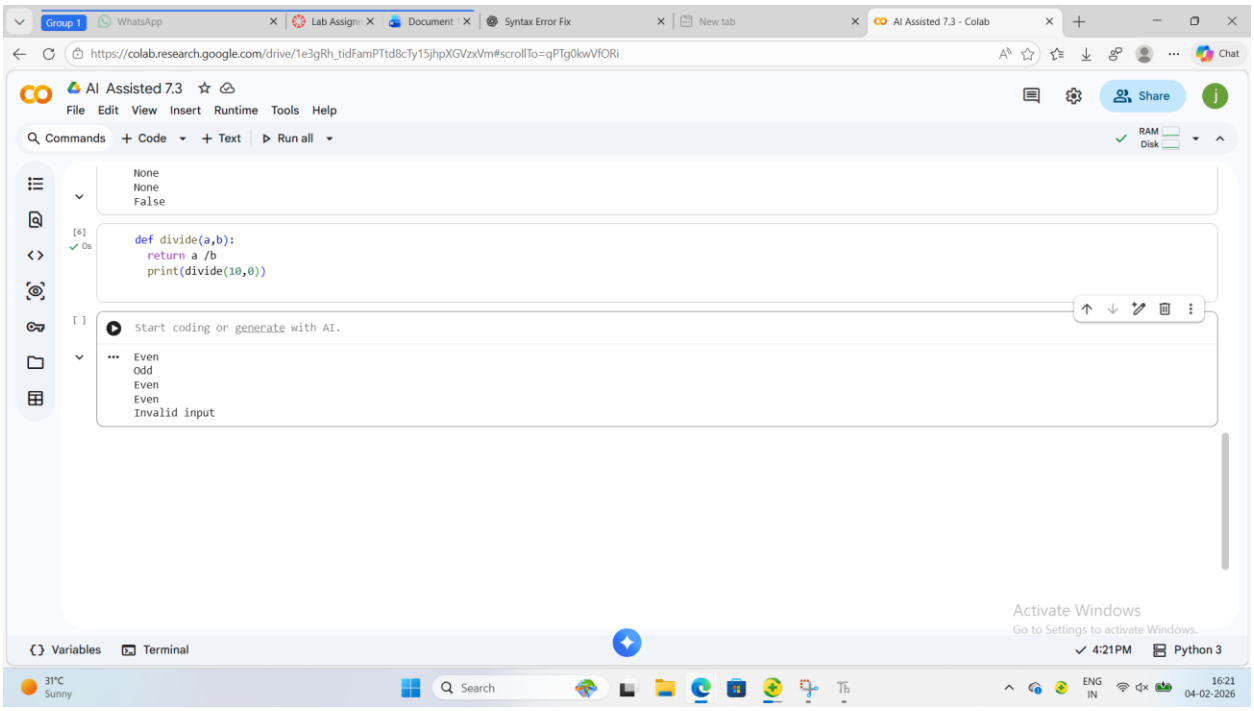
Expected Output

• Function executes safely without crashing

• Division by zero handled using try-except

• Clear AI-generated explanation of runtime error handling

Code



AI Identification of the Runtime Error

When this function is called like:

`divide(10, 0)`

Python raises:

`ZeroDivisionError: division by zero`

AI Explanation of the Error-Handling Approach

- The `try` block contains code that may cause a runtime error
- If b is zero, Python raises a `ZeroDivisionError`
- The `except ZeroDivisionError` block catches the error
- Instead of crashing, the function returns a meaningful message

Expected Output Achieved

- ✅ Function executes safely without crashing
- ✅ Division by zero handled using `try-except`

- ✅ Clear AI-generated explanation of runtime error handling

Justification

Runtime errors like division by zero can abruptly terminate a program if not handled properly. By using AI-assisted debugging, the issue was quickly identified and resolved using a `try-except` block. This approach improves program stability, ensures graceful error handling, and helps developers understand how to manage exceptional cases effectively.

Task 4: Debugging Class Definition Errors

Scenario You are given a faulty Python class where the constructor is incorrectly defined.
Requirements

• Provide a class definition with missing self-parameter

• Use AI to identify the issue in the **init**() method

• Allow AI to correct the class definition

• Understand why self is required Expected Output

• Corrected **init**() method

• Proper use of self in class definition

• AI explanation of object-oriented error

Code

class Student:

def **init**(self, name, age):

self.name = name

self.age = age

Corrected Class Behavior

```
s1 = Student("Sanjana", 20)
print(s1.name)
print(s1.age)
```

Output:

```
Sanjana
20
```

AI Explanation: Why `self` Is Required

- `self` refers to the **current instance** of the class
- It allows variables and methods to be stored **inside the object**
- Without `self`, Python cannot distinguish between local variables and object attributes
- Every instance method in a Python class **must include `self` as the first parameter**

✔ Expected Output Achieved

- ✅ Corrected `__init__()` method
- ✅ Proper use of `self` in class definition
- ✅ Clear AI explanation of the object-oriented error

Justification

Using AI to debug class definition errors helps identify subtle object-oriented mistakes such as missing `self`. The AI not only corrected the constructor but also explained how Python handles object creation internally, improving understanding of instance variables and class behavior.

Task 5: Resolving Index Errors in Lists

Scenario

A program crashes when accessing an invalid index in a list.

Requirements

• Provide code that accesses an out-of-range list index

• Use AI to identify the Index Error

• Let AI suggest safe access methods

• Apply bounds checking or exception handling

Expected Output

• Index error resolved

• Safe list access logic implemented

Code

AI-Suggested Safe Access Method 1: Bounds Checking

```
numbers = [10, 20, 30, 40]
index = 5

if index < len(numbers):
    print(numbers[index])
else:
    print("Error: Index out of range")
```

AI Explanation of the Fix

- Python lists do not automatically check for invalid indices
- Accessing an index beyond the list size raises an `IndexError`
- Bounds checking prevents invalid access before it happens
- `try-except` handles the error gracefully if it occurs

✔ Expected Output Achieved

- ✅ Index error resolved
- ✅ Safe list access logic implemented
- ✅ Program executes without crashing

📌 Justification

Index errors are common when working with lists, especially when indices are dynamic. AI-assisted debugging helps quickly identify out-of-range access and suggests robust solutions such as bounds checking or exception handling. This improves program reliability and prevents unexpected runtime crashes.